UCRL-PROP-220408

PSAAP Computer Science Research Issues

David Jefferson, Bronis de Supinski, Michael Zika (LLNL)
Marvin Alme (LANL)
Daniel Rintoul (Sandia)

31 March 2006

The purpose of this white paper is to suggest research areas in computer science that can contribute directly to the advancement of the "Predictive Science". Whereas the companion white papers tend to focus on specific application areas of interest to the ASC effort, here we specifically invite computer science researchers to think broadly about technologies that enable complex, scalable simulations, especially with emphasis on methodologies for verification and validation. The research we solicit need not maintain continuity with current practices, but instead may represent dramatic departures. We encourage research motivated by the question "How could we advance the science of prediction if we were given a clean slate, with the freedom to re-invent scientific computation?"

The following is an outline of potential research areas in computer science that could support the Predictive Science Academic Alliance Program (PSAAP). It is not to be read prescriptively; the topics are simply illustrative of some of the directions that might profitably be pursued. The **major headings** refers to important areas of interest, while the minor bullets flesh out some possible approaches. But other issues or approaches not listed are welcome, and may be superior.

Successful proposals will not treat these as independent computer science research areas, but will specifically connect them to improvement in our simulation capability in the required science and engineering disciplines. A proposal should not only discuss the computer science research intended, but also the improvement in simulation technology and predictive science expected, and how it will be demonstrated.

New scalable algorithms

New, scalable parallel algorithms, both at the application level and systems levels, are essential to progress in ASC applications. Other white papers cover the algorithmic needs of some specific application areas. Of interest here are algorithms that are not covered elsewhere, and that are novel in some fundamental way, or cut across many application areas, e.g. algorithms that

- have important provable properties, e.g. deadlock- and starvation-freedom, robustness against faults, lower time, space, or bandwidth complexity (worst case, average case, or probabilistic) than previous comparable algorithms, or better numeric or physics properties (faster convergence, tighter error estimators, better energy conservation);

- are MPMD, self-balancing, asynchronous, use interval arithmetic or automatic differentiation, are fault-avoiding, speculative, probabilistic, or otherwise outside the current programming paradigms, especially if they can also be made portable and/or deterministic;
- solve problems in the OS, I/O system, runtime system, communication library, compiler, optimizer, fault tolerance systems, or other systems issues outside of the application code;
- exhibit scalability to the level of 100,000 processors or more, to fully utilize the capability architectures, both existing and planned.

Algorithms and programming technology specific to parallel simulation

Simulations are a distinct category of software object—a hybrid of computation, geometry, and physics. They have special properties that distinguish them from all other computations, notably the special logical status given to simulation time as a temporal coordinate system that must be consistent with causality, along with one or more dimensions of simulation space as well. Parallel simulations are particularly challenging because of the need for proper synchronization and load balancing, and the potential value of speculative/optimistic methods. In the long run, we need simulations to be able to be built not as standalone programs, but packaged and modularized to serve as components of larger systems. The research issues specifically designed to support scalable simulations include these:

- componentization and formal interfaces technologies designed for simulations, giving logical status to time and space coordinate systems, and to geometry in general;
    - should allow a simulation to be queried and controlled from external code;
    - should extend to files, databases, and I/O associated with simulations;
- theory of, and algorithms for, coupling simulations spatially, temporally, at multiple scales, in ensembles, etc.;
- methods for coupling simulations to non-simulation components, e.g. visualizers, or databases (especially temporal and geometric databases);
- unification of methods for the parallel simulation of discrete, continuous, and mixed systems, and optimistic methods for continuous simulation;
- physical units (meters, kilograms) incorporated into the type systems of programming languages, along with compile-time and/or runtime type checking extended to units checking, and automatic translation/coercion from one set of units or coordinates to another;
- domain-specific language constructs, e.g. improved support for operators such as grad, curl, etc. on large matrices; or tensor operators;
- efficient execution of simulations with components that have disparate and time-varying length and/or time scales;
- techniques for solving PDEs or mixed discrete and continuous systems using fully unstructured 4-d (3 space + time) meshes, i.e. variable time steps and arbitrary time-varying meshes, especially if extendable to even higher dimensional unstructured meshes.

New parallel programming models

Parallelism occurs at all levels of memory hierarchy and all architectural scales, with associated styles of communication and synchronization at each:

- pipelining, multi-issue, vectors, microthreading (shared registers, cache)
- sequential threads (locks, shared memory, cache)
- SPMD processes (OpenMP, MPI, shared files)
- MPMD components, (asynchronous messages, MPI, RPC, RMI, shared files)
- wide-area (internet, grid) components (TCP/IP, Globus, etc.)

We need parallel programming paradigms that cope in an organized way with this enormous (nine orders of magnitude) range of scales, from nanoseconds to seconds, and a comparable range of data object sizes. Among the paradigmatic issues that occur at more than one level are these:

- nestable and composable parallel software abstractions, e.g. parallel objects (class instances) of various kinds;
- componentization (parallel composition of, and communication between, separately-developed codes);
- migratable units of computation (for load balancing, latency minimization, and fault avoidance);
- support for checkpoint/restart, replication, or retry at the object, process, or component levels, or other fault-management mechanisms;
- parallel communication primitives, e.g. parallel method invocation from one parallel computation to another;
- speculative and optimistic algorithms and synchronization, perhaps based on technology for parallel rollback, reverse computation, journaling, transaction abortion, etc.;
- parallel instrumentation, optimization, and debugging technology;
- new language, interpreter, compiler, and/or library support for all of the above;
- new software build technologies that are more reliable, less error prone, parallel, or otherwise improved

Parallel componentization technology

Large parallel codes, especially simulations, should not generally be standalone programs. They should be composed from, and subsequently packaged as, reusable components. A large body of research has been done on componentization, but much more is in progress and needs to be done. For computation and simulation to fulfill their promise as the "third leg of the stool" of the scientific method, alongside theory and experiment, we need to be able to create documented, reusable libraries of codes that can be used in various ways as modules in larger computations. We need research along the following lines (among others):

- component technologies that allow components to be dynamically launched and executed in parallel;
- nested (recursive) composition and invocation of components;

- language-independent formal interfaces between components that go beyond APIs, to include information in addition to method signatures, e.g. mesh and simulation time information, physical units, etc.;
- component introspection;
- component migration;
- component checkpoint/restart;
- parallel and collective inter-component communication primitives, e.g. the "MxN" problem.

Software fault detection/avoidance/recovery technology

All scalable software needs to be defined and engineered with fault management in mind. For example, are RAID-like strategies possible at the algorithmic/application level, i.e. can we create algorithms that have sufficient internal redundancy to detect failures, but that are much more efficient than full replication and comparison of outputs? Fault tolerance strategies are needed in:

- *algorithms and applications:* Can the algorithms themselves be coded redundantly and efficiently to be robust against faults?
- *parallel programming languages:* What linguistic facilities can play a role in fault detection or recovery, e.g. checkpoints as first class objects, compiler support for checkpoint and/or rollback, or compiler-aided replication?
- *operating systems:* Can the OS support migration away from faults, or make the state the OS holds on behalf of user applications more visible and accessible for checkpoint and recovery operations?
- *communication packages:* Can they route around faults, or deal transparently with senders and receivers that have migrated away from faults? Can they support checkpoint/restart without bringing the entire application to a global synch point?
- *software libraries:* Can libraries that hold hidden state make it visible in a structured way for use in checkpoint, restart, and migration?
- *component technologies:* Can componentized applications recover from the failure of a single component by methods that do not affect the other components?

Operating system support for capability and capacity machines

Advanced simulation technologies will likely depend on scalable operating system advances to address issues such as fault tolerance, componentization, optimistic synchronization, load balancing, scalable I/O, and new models of computation. The PSAAP program may support operating system research if it is specifically tied to the needs of complex ASC simulations. Topics such as the following issues might be appropriate if such a case is made:

- the ability to boot different OSs in different partitions, so that applications needing different OSs can run concurrently on capacity machines;
- scalable OS boot, job launch, and dynamic library load mechanisms;
- efficient support for transparent load migration for load balancing, fault avoidance, and node compaction;

- dynamic node allocation and freeing for expanding and contracting jobs (and associated programming model support and suballocation of nodes *within* the job) for component launch;
- collective system calls;
- support for one-sided, interrupting, message communication primitives;
- efficient gang scheduling, including preemptive priority gang scheduling.

Scalable parallel I/O technology and abstractions

Parallel I/O research seems always to lag other areas in parallel computation, but many applications on the horizon are likely to be dominated by I/O issues. We need new technology for:

- parallel file systems and associated synchronization;
- parallel relational databases;
- parallel geometric and temporal databases;
- parallel input from sensor networks or arrays, including asynchronous and real time input;
- parallel output to visualization systems.

Acknowledgement

This work was, in part, performed by the Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratories under auspices of the U.S. Department of Energy.
